An overview of gradient descent optimization algorithms

> An overview of gradient descent optimization algorithms
> Sebastian Ruder, 2016

Reviewer : 박이현

Introduction

Gradient Descent는 신경망 최적화 과정에서 많이 사용되는 최적화 기법이다. 따라서 딥러닝 분야에서 많이 활용되고, 이를 기반으로 다양한 최적화 모델이 개발되었다. 이 논문에서는 다양한 GD 기반의 최적화 기법들에 대해서 간단하게 다루고 있다.

Gradient Descent에 대해서 간단히 소개하면, 특정 파라미터에 대한 목적함수 $J(\theta)$를 최소화하는 방법이고 주로 목적함수를 파라미터 θ에 대해 1차 편미분을 진행하고, 해당 값인 $\nabla_\theta J(\theta)$의 반대방향으로 step의 크기를 결정하는 학습률(learning rate) η를 곱해서 원래 파라미터에서 빼는 방식으로 파라미터를 최소값을 향해 계속 업데이트한다.

Gradient descent variants

GD 기법을 이용한 3가지 방법론에 대해서 먼저 한번 간단하게 알아보려고 한다. 위에서 제시된 GD의 방법을 이용한다는 것은 같지만, 목적함수의 미분을 위해 얼마나 많은 데이터를 사용해 계산하는 지가 다르다.

1) Batch Gradient descent

Batch Gradient descent는 가장 기본적인 GD의 일종으로, Vanila Gradient descent라고도 한다. 파라미터는 다음과 같이 업데이트된다.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

BGD 방법에서는 한번의 파라미터 업데이트를 위해서 전체 데이터에 대해 모두 Gradient를 계산하고 평균을 구한다. 이후 그 값에 학습률을 곱해서 반대방향으로 이동함으로써 최적의 파라미터값을 찾아간다. 해당 방법은 convex한 형태의 목적함수에 대해서는 Global

minimum으로의 수렴을 보장하지만, 그에 따른 Trade-off로써 속도가 매우 느리다. 또한 Large Dataset에 대해서는 메모리 문제도 발생하게 된다.

2) Stochastic Gradient descent

확률적 경사하강법인 Stochastic Gradient descent는 기존 BGD의 느린 속도를 보완하기 위한 방법으로, 파라미터 업데이트 과정은 아래와 같다.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

각각의 파라미터 업데이트 과정에서 하나의 데이터만 선택하여 해당 데이터의 Gradient를 계산하고, 이를 학습률과 곱해 이동함으로써 최적의 파라미터 값을 찾게 된다. 전체 데이터에 대한 기울기를 모두 고려할 필요가 없기 때문에 BGD보다 훨씬 빠르게, 자주 파라미터의 업데이트를 진행한다.

하지만 빠른 만큼 데이터를 하나만 선택하기 때문에 Variance가 높아진다는 단점이 존재하며, 이에 따라 목적함수가 심하게 변동할 수 있다.

BGD는 모든 데이터를 고려하기 때문에 대체로 Global minimum을 향해서 간다면, SGD는 Local minimum에 빠질 가능성이 있다. 또한 정확한 minimum 값에 근접하기도 어렵다. 이러한 문제를 해결하기 위해 학습률을 점차 낮추면 BGD와 유사한 결과를 낼 수 있다.

3) Mini-batch Gradient descent

SGD의 큰 변동성과 BGD의 속도 문제를 모두 고려하기 위해, Mini-batch Gradient descent라는 방법이 등장하였다. 파라미터 업데이트는 다음과 같다.

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

기본적인 방식은 SGD와 같으나 한 개의 데이터만 이용하는 것이 아닌, n개의 데이터를 가진 작은 batch를 만들어서 해당 batch 내에 존재하는 데이터들의 gradient 평균을 통해 파라미터를 업데이트한다.

이를 통해 다음과 같은 효과를 얻을 수 있다:

1) 파라미터의 변동성(Variance) 감소

2) 여러 개의 데이터를 이용함으로써 벡터화를 시켜 적용 가능(딥러닝에 필요)

일반적으로 batch의 크기는 50~256으로 설정되지만 다양한 수치로 적용될 수 있고, 신경망 학습에 MGD는 자주 사용이 된다. SGD도 이런 관점에서 본다면 batch의 크기가 1인 MGD 방법론이라고 할 수 있다.

Challenges

일반적으로 많이 사용되는 MGD 방법은 반드시 좋은 최적화값을 보장해주지는 않는다. 그 이유는 아래와 같은 문제들이 존재하기 때문이다.

- **적절한 학습률의 사전 설정이 어려움**

  학습률이 작을 시, 최적화 속도가 매우 느리고 너무 크다면 최적값 근처에서 수렴하지 못하고 지속적으로 변동할 수 있다.

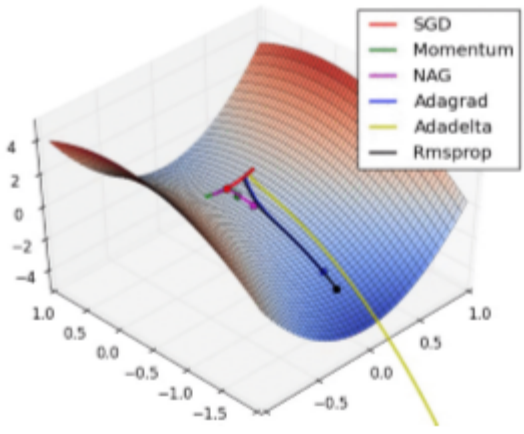
  해당 문제를 해결하기 위해서 제시할 수 있는 방법으로는 학습률을 미리 정해진 시점부터 줄여나가는 방식을 통해서 어느정도 minimum에 근접하게 되면 최적값 근방에서 변동을 줄일 수 있다. 하지만 작동 중에는 변경할 수 없으므로 사전에 선택되어야 하며, 데이터의 특성을 명확하기 파악하지 못했을 경우에는 임계치 설정이 어렵다.

- **모든 파라미터에 동일한 학습률이 적용됨**

  만약 데이터 자체가 sparse하거나, 변수들 간의 변화 정도가 다르다면 우리는 변수별로 다른 학습률을 적용해야 한다. 하지만 GD에 적용되는 학습률은 모든 파라미터 θ에 대해 η로 동일하다. 따라서 거의 존재하지 않는 변수에 대해서도 많은 파라미터 업데이트를 진행하게 되므로 낭비가 발생한다.

- **non-convex한 목적함수에 대한 Trap 현상**

  convex하지 않은 목적함수에 대해서 GD를 적용하게 되면, 수많은 local minimum이 존재할 가능성이 높아진다. 하지만 GD 방법은 기울기의 반대방향으로 파라미터를 업데이트하므로 한번 local minimum에 빠지면 갇히게 될 수 있다. 또한 saddle point(안장점) 지역 일대에서는 기울기가 대부분 0에 매우 근사하기 때문에 탈출이 어렵다.



(b) SGD optimization on saddle point

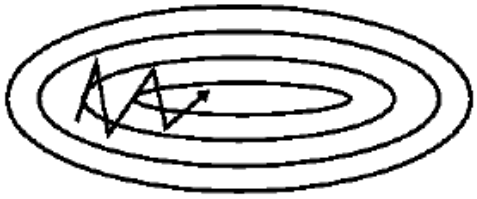
Gradient descent optimization algorithms

기존의 BGD, SGD, MGD 알고리즘은 위에서 제시한 문제들을 내포하고 있다. 따라서 이를 해결하기 위해 다양한 기법들이 개발되어왔는데, 하나씩 소개해보고자 한다. 고차원 데이터 셋에 대해 다루는 Newton's method와 같은 방법은 제외하였다.

1) Momentum

모멘텀은 문자 그대로 해석하면 가속도를 의미하고, 직관적으로 이해하면 진행 방향에 대해 관성을 부여하는 것이다. 앞에서 제시되었던 문제들 중 local minimum에 빠지는 문제를 해결할 수 있는 방법으로 제시되었다.

(a) SGD without momentum



(b) SGD with momentum

모멘텀은 최적화가 진행되는 방향으로 SGD가 진행되는 것을 가속화시키며, 그렇기 때문에 최적값 근처에서 진동하는 현상을 어느 정도 잡아줄 수 있게 된다. 모멘텀에서는 기존의 SGD에서 이전 시점의 벡터에 대한 정보를 일부분 활용하게 된다.

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

일반적으로 γ는 0.9 혹은 유사한 값으로 설정이 되며, 이전 벡터의 90%를 모멘텀에 대한 정보로써 이용하게 된다. γ는 모든 차원이 gradient를 같은 방향으로 가리킬 경우 증가하게 되며, gradient가 반대 방향으로 방향을 바꾸면 감소한다.

즉 내리막길과 같이 모든 gradient가 아래 방향을 가리킬 경우 내려갈 거리가 아직 많이 있기 때문에 해당 방향으로 가속도를 증가시키고, gradient가 반대방향으로 바뀌었다는 것은 오르막길에 접어들었다는 것을 의미하기 때문에 가속도를 줄여 최적점을 찾는다.

모멘텀을 활용하면, 우리는 기존 SGD보다 더 빠른 수렴속도와 적은 변동성을 갖게 된다.

2) Nesterov accelerated gradient

Nesterov accelerated gradient(NAG)는 모멘텀을 보완한 모델로써, gradient가 바뀌는 지점까지 최적화가 진행되기 이전에 최적점을 인식하고 속도 및 위치를 조절하는 알고리즘이다.

일종의 예지 능력을 갖춘 모멘텀 알고리즘이며, 아래와 같이 작동한다. 모멘텀과 매우 유사한 형태를 띄게 된다.

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

현재 파라미터 위치에서 현재의 모멘텀을 적용하면 새로 업데이트 되어 위치할 다음 파라미터 위치를 계산한다. 하지만 현재의 gradient인 v_t를 알 수 없기 때문에 그와 근사한 v_{t-1}를 사용해서 모멘텀만큼을 적용하여 목적함수에 적용시켜준다. 그리고 학습률을 반영하여 모멘텀을 계산한 후 최종적으로 파라미터를 업데이트 하게 된다.

기존 벡터의 위치에 대한 정보를 일부분 활용하면서, 향후 위치에 대해서 사전에 목적함수에 대한 계산을 진행한다. 해당 과정에서 gradient가 일부 소실이 되나, 향후 파라미터 값을 예측할 수 있게 되므로 SGD의 fluctuation 문제를 해결할 수 있다.

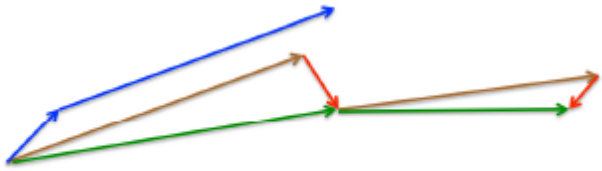


Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

파란색 벡터는 모멘텀을 의미하며, 모멘텀의 원리에 의해서 gradient와 가장 연관성이 높은 방향으로 최적화가 진행되게 된다. NAG는 **갈색 벡터**로, 모멘텀의 방향으로 최적화를 우선 진행한 뒤, 향후에 존재하게 될 위치에 대한 계산을 진행한다. 그에 맞게 **빨간색 벡터**만큼 자신의 위치를 수정하게 되고, 따라서 **초록색 벡터**로 변하게 된다.

NAG를 통해 우리는 모멘텀의 속도를 조절하고, RNN 모델에서 높은 반응성을 보일 수 있게 된다. 그리고 목적함수의 기울기를 최적화 과정에 반영할 수 있게 되며, 이를 통해 빠른 SGD의 진행을 돕는다.

또한 파라미터의 중요도에 따라 다른 업데이트 정도를 갖게 되므로 변수 중요도에도 이용될 수 있다.

3) Adagrad

Adagrad는 Adaptive gradient의 약자로써, 많이 변화한 파라미터에 대해서는 적게 변화하는 방향으로 학습률을 낮추게 되며, 변화량이 적은 파라미터에 대해서는 학습률을 높이게 된다. 이러한 특성으로 인해 sparse한 데이터에 대해서 최적화가 잘 작동한다. 모든 파라미터가 최적점으로 가기 위해서는 자주 업데이트 되지 않는 파라미터들은 한번 업데이트 할 때 큰 폭으로 움직여야 할 것이며, 자주 업데이트가 된다면 작은 폭으로 움직여야 한다. 이런 원리를 활용한다.

Adagrad를 통해 우리는 대용량 신경망에 대해 SGD보다 더 robust한 최적화를 진행할 수 있으며, 파라미터에 대한 선택적 학습률 적용을 통해 단어 임베딩 등의 자연어 처리에도 탁월한 효과를 가져왔다.

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$
$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

앞에 제시된 모델들은 모두 파라미터 θ에 대해 한번에 업데이트 하는 방식으로 진행되었지만, Adagrad는 개별 파라미터 θ_i에 대해 반복을 진행할 때마다 다른 학습률을 적용한다. $J(\theta_{t,i})$는 개별 파라미터에 대한 t시점의 목적함수를 의미한다.

SGD는 θ_i에 대해 매번 업데이트를 각각 진행하고, 이 과정에서 기존 시점의 파라미터를 참고하여 학습률을 수정한다.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_{t,i}$$
$$G_{t,ii} = \sum_{i=1}^{t} (\frac{\partial J(\theta_i)}{\partial \theta_i})^2$$

G_t는 개별 파라미터에 대한 gradient를 제곱한 합의 Diagonal Matrix를 의미한다. 즉 $\theta_1, \theta_2, \cdots$에 대한 t시점까의 gradient 제곱값의 합이다. ϵ은 분모가 0이 되지 않기 위한 smoothing term 역할을 하며, 주로 $1e-8$로 설정한다.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$

이렇게 개별 파라미터에 대한 수정된 학습률에 행간 원소별 곱셈을 통해 표현한 식은 위와 같으며, $\odot$는 아다마르 곱셈이다.

▼ 아다마르 곱셈(Element-wise matrix-vector multiplication)

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \end{bmatrix}$$

Adagrad를 통해 우리는 학습률을 자동으로 조절할 수 있게 되며, 0.01을 default 값으로 일반적으로 설정한다.

Adagrad는 단점이 존재하는데, 학습이 지속될수록 G의 값이 계속적으로 더해지기 때문에 분모가 ∞로 발산할 수 있다. 그럴 경우 학습률에 대한 term이 0에 수렴하기 때문에 더이상 파라미터가 업데이트 되지 않게 된다. 이러한 단점을 해결하기 위해 아래의 Adadelta, RMSprop 모델이 등장하게 되었다.

4) Adadelta

Adadelta는 위의 학습률 소실 문제를 해결하기 위해서 제시된 모델이다. 매 시점의 gradient를 모두 합하는 대신, 고정된 w시점만큼의 과거 gradient만을 참고한다.

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

Adagrad와 다르게 gradient 제곱의 합이 아니라 기댓값(평균)인 $E[g^2]_t$를 사용하게 됨으로써 우리는 값이 발산하는 것을 방지할 수 있게 된다.

$t-1$시점까지 쌓였던 gradient의 제곱을 평균을 낸 후, 현 시점의 gradient의 제곱과 가중평균(지수평균)을 내어 새롭게 G_t를 업데이트하게 된다. 여기서 γ는 모멘텀과 유사한 0.9 정도로 설정이 되며, 이에 따라 업데이트에 관한 식을 다시 표현하면 아래와 같다.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t = \theta_t - \Delta\theta_t$$
$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t = -\frac{\eta}{RMS[g]_t} \cdot g_t$$

위와 같이 $E[g^2]_t + \epsilon$를 $RMS[g]_t$로 바꾸어서 표현할 수 있는데, 그 이유는 RMS(Root Mean Square)가 아래와 같이 표현될 수 있기 때문이다. gradient의 기댓값은 평균과 같은 의미이고, 그렇기 때문에 RMS로 표현한다.

$$RMS[g]_t = \sqrt{\frac{g_1^2 + g_2^2 + \cdots + g_n^2}{n}}$$

이 때, θ_t의 변화량을 앞에서 했던 것과 같이 이전 시점까지의 평균과 현재 시점에 대한 변화량의 가중평균으로 설정하고, 이를 RMSE의 관점에서 다시 식을 쓰면 아래와 같다.

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)g\Delta\theta_t^2$$
$$\theta_{t+1} = \theta_t - \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$

우리는 t시점의 $\Delta\theta$에 대한 RMSE를 알 수 없기 때문에, 대략적으로 그와 유사한 $t-1$시점의 RMSE 값으로 추정을 진행한다. 파라미터의 RMSE를 통해서 학습률을 자동으로 추정할 수 있기 때문에 우리는 굳이 초기 학습률을 설정해주지 않아도 된다.

RMSprop도 Adadelta와 동일한 원리로 적용이 되므로 여기에서는 설명을 생략한다. 이와 같이 Adadelta를 통해서 우리는 과거의 gradient들의 영향력을 줄일 수 있고, 이를 통해 학습률이 소실되어 업데이트가 더 이상 되지 않는 Adagrad의 문제를 해결한다.

5) Adam

Adam은 Adaptive Moment Estimation의 약자로, 각 파라미터에 대해 학습률을 유연하게 적용하는 알고리즘이라는 점에서 Adadelta와 RMSprop과 비슷하다. 또한 과거의 gradient의 제곱에 대한 영향력을 줄여준 다는 점에서도 동일하다. Adam은 여기에 더해 과거 gradient를 가중평균의 형태로 참고하게 되는데, 앞에서 봤던 모멘텀의 개념이 적용된 것이다. Adagrad + Momentum이라고 보면 될 것 같다.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

m_t는 gradient의 1차 적률에 대한 추정값이며, v_t는 2차 적률에 대한 추정값이다. 1차 적률은 $E(X)$이고, 2차 적률은 $E(X^2)$인데, 이를 추정한 값을 Adam에서는 각각 m_t, v_t로 표현한다. 이전 시점의 적률과 현재 시점의 기울기의 가중평균을 내어 새롭게 업데이트를 진행한다. Adam은 초기 상태이거나, β_1, β_2가 아직 1에 근사해서 이전 시점의 gradient들의 영향력이 클 때, m_t, v_t는 0으로 편향된다. 따라서 아래와 같은 식으로 편향을 조절해준다.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

이렇게 기존 목적함수에 대해 현재와 과거의 가중평균을 이용하여 Adagrad와 동일한 방식으로 학습률을 각 파라미터에 맞게 자동적으로 적용시킨다. 업데이트를 시키면 아래와 같이 파라미터가 나오게 된다.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

즉, 기존 목적함수에 대한 기울기는 모멘텀의 개념에서 차용해서 이전 시점의 벡터가 가리켰던 방향으로 진행하게 되며, 학습률에 대해서는 Adadelta의 개념에서 차용하여 기존

gradient들의 제곱을 참고하여 변수별로 다르게 학습률을 적용시키게 되는 알고리즘이라고 생각하면 편하다.

일반적으로 default 값은 $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$이다.

6) AdaMax

AdaMax는 Adam과 비슷한 모형이며, 차이점은 기존의 Adam은 v_t에 대해 ℓ_2 norm을 적용해서 업데이트를 진행해 준 반면에 AdaMax는 ℓ_p norm을 적용해주었다는 점이다.

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p) \left| g_t \right|^p$$

일반적으로 ℓ_p norm은 불안정해지는 특징이 있지만, 만약 $p \to \infty$의 경우에는 안정적인 성능을 보여준다. 이러한 이유로 AdaMax는 Adam보다 더 안정된 값을 제공하기도 한다. AdaMax의 파라미터 업데이트 과정은 아래와 같다.

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) \left| g_t \right|^\infty \\ &= max(\beta_2 \cdot v_{t-1}, \left| g_t \right|) \\ \theta_{t+1} &= \theta_t - \frac{\eta}{u_t} \hat{m}_t \end{aligned}$$

u_t는 max 함수에 의존하기 때문에 Adam과 같이 m_t, v_t가 0으로 편향된다는 단점이 적다. 그래서 별도의 bias correction을 진행해주지 않아도 된다. 좋은 Default 값은 일반적으로 $\eta = 0.002, \beta_1 = 0.9, \beta_2 = 0.999$이다.

7) Nadam

Adam은 모멘텀과 RMSprop(or Adadelta) 방식의 혼합이라고 할 수 있다. RMSprop을 통해서 기존 gradient 제곱의 영향력을 지속적으로 감소시키며, 모멘텀을 통해서 이전 gradient의 영향력을 반영한다.

Nadam은 앞에서 제시된 NAG 방법과 Adam 방법의 혼합이다. NAG를 통해서 우리는 gradient를 구하기 전에 모멘텀을 진행해서 해당 방향으로 이동한 후, Adam을 통해 각각 파라미터를 업데이트해주는 방식이다. Adam 알고리즘을 다시 한번 확인해보자.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned}$$

Adam에서 이용한 모멘텀 방식은 이전 시점의 모멘텀인 m_{t-1}을 이용했지만, 이제는 NAG의 원리에 의해 앞의 시점을 예측하기 때문에 m_t를 이용해야한다. 따라서 우리는 과거 시점에 대한 모멘텀을 현재 모멘텀으로 근사시켜준다.

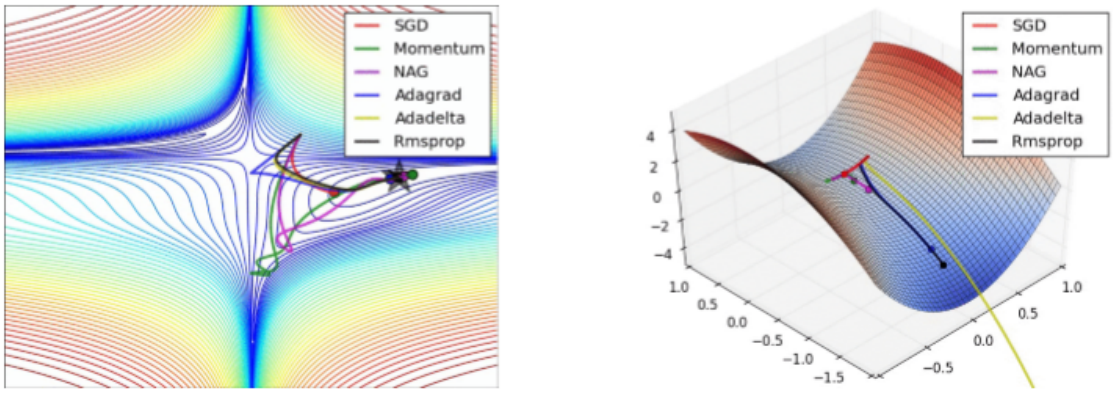
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t})$$

Adam에서는 이전 시점의 벡터와 현재 시점의 벡터를 가중평균을 해주었고, 따라서 그에 따른 편향값을 조정해주기 위해서 $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$를 적용해주었다. 하지만 Nadam에서는 이전 시점에 대한 편향 조정이 필요가 없다. 따라서 이전 모멘텀에 대한 편향 조절을 생략하고, 이를 t 시점의 모멘텀 벡터로 근사시킨다. Nadam의 최종 식은 아래와 같다.

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}(\beta_1 \hat{m}_t + \frac{(1-\beta_1)g_t}{1-\beta_1^t})$$

알고리즘 별 시각화

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html



(a) SGD optimization on loss surface contours (b) SGD optimization on saddle point

Figure 4: Source and full animations: Alec Radford

위에서 제시된 알고리즘 별로 시각화를 진행해보면, saddle point에서의 성능을 확인할 수 있다. Adagrad, Adadelta, RMSprop은 바로 최적점을 찾아서 나가는 반면, 모멘텀과 NAG는 갇혀서 탈출하지 못하는 모습을 보인다.

일반적으로, 이런 상황에서는 Adagrad, Adadelta, RMSprop, Adam과 같은 adaptive learning-rate 방식이 최적의 수렴성을 제공한다.

어떤 Optimizer를 사용해야 하는가?

특정 상황에서 어떤 Optimizer를 사용해야 하는지에 대한 정답은 없지만, 논문에는 몇몇 특정한 상황에서 일반적으로 잘 작동하는 최적화 알고리즘을 제시하였다.

- 데이터가 Sparse한 경우 : Adagrad, Adadelta, Adam 등 adaptive learning-rate 기반 방법론
- RMSprop, Adagrad, Adadelta 등의 알고리즘에는 존재하지 않는 bias-correction 으로 인해 일반적으로 Adam이 가장 좋은 성능을 보임

최근 다양한 논문에서는 모멘텀을 이용하지 않고, 간단한 학습률을 적용한 일반 SGD를 사용하는 경우가 종종 있다. 그 이유는 Vanilla SGD가 비록 시간은 다른 Optimizer에 비해 오래 걸리지만 일반적으로 최소값을 robust하게 잘 찾기 때문이다. 또한 saddle point에서는 탈출하지 못하는 결과를 보이지만, local minima에서는 잘 탈출할 수 있다는 이유도 있다.

결과적으로, 빠른 수렴성과 복잡한 인공신경망 모델에서는 Adaptive learning-rate 기반 방법론을 쓰는 것이 좋다.

Parallelizing and Distributing SGD

일반적으로, 군집화 정도가 낮고 대용량인 데이터를 다루기 위해서는 Parallelizing과 distributing을 진행해주는 것이 좋다. (적당한 번역을 못찾겠음..) 아래와 같은 다양한 프로그램이나 알고리즘을 이용하면 좋은 SGD 성능을 낼 수 있다고 언급된다. 참고만 하자.

- **Hogwild!**

  CPU에서 병렬화를 통해 SGD를 진행할 수 있게 작동하며, 데이터가 sparse한 경우에만 작동한다.

- **Downpour SGD**

  비동시적인 SGD의 방법으로써, Training Data에 대해서 병렬적인 복제 모델을 여러 개 만들어내는 방식으로 작동한다. 각 모델에서 업데이트 된 파라미터는 서버로 보내지며, 파라미터의 부분을 맡아서 업데이트를 진행한다. 하지만 각 모델끼리는 서로 간섭할 수 없기 때문에 수렴이 이루어지지 않고 발산할 수 있다는 가능성이 존재한다.

- **Delay-tolerant Algorithms for SGD**
- **TensorFlow**
- **Elastic Averaging SGD**

Additional strategies for optimizing SGD

SGD의 성능을 높이기 위한 여러 방법들을 논문에서는 추가적으로 제시한다.

- **Shuffling and Curriculum Learning**

  최적화 알고리즘에서는 편향 문제가 발생할 수 있기 때문에, 이를 해결해주는 방법은 매 반복 시마다 training data를 섞어주는 것이 좋다.

  하지만 특정 상황에는 데이터가 의미있는 순서로 정렬하는 것이 더 좋은 결과를 낼 수도 있다. 이렇게 정렬하는 방식을 Curriculum Learning이라고 한다.

- **Batch normalization**

  평균이 0이고 분산이 1이 되도록 데이터를 정규화해주는 것이 좋다. 일반적으로 Adam과 같은 최적화 알고리즘에서는 파라미터들에 대해 다른 정도로 업데이트를 진행해주는데, 이렇게 되면 한번 업데이트 할 때마다 정규성이 깨지게 된다. 따라서 batch normalization을 이용하는 것이 좋다.

- **Early stopping**

- **Gradient noise**

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2)$$

  개별 gradient에 noise를 추가해줌으로써 새로운 local minima를 통해 탈출할 수 있도록 모델을 도와준다.